Lab 10: String Operations

Taaruni Ananya | Session 008 | CRN: 15623

04/11/2025

Dr. Lan Gao

Introduction

Lab 11 is centered around programming for taking strings and processing them, which involves practicing writing code with RISC-V, VSCode, and Terminal as well as practicing using the ASCII values. Lab 11 looks into exploring how to use loops to write a program that can take in a string, count the number of characters in it, and return an output for count. The lab explains how to use Terminal, VSCode, and RISC-V. The lab also gives hands-on experience with writing a program to practice processing strings to perform certain operations such as comparing, counting, and looping through.

Purpose

The purpose of this lab is to learn how to program efficiently as well as the use of loops for characters in strings, along with the commands and practice writing RISC-V utilizing VSCode; this lab also works to use strings and loops to find character count as well as string comparison using ASCII values. This lab is meant to experiment with using RISC-V to take in strings and return output with programs and explore different ways to use these commands.

Main Objectives

- Learn how to copy a string
- Learn how to count the number of occurrences of a character in a string
- Learn how to compare two strings
- Learn how to use ASCII values
- Reinforce learning of bne, beq, and branching operations in Assembly
- Reinforce learning of loops
- Learn to process strings

Brief Explanation

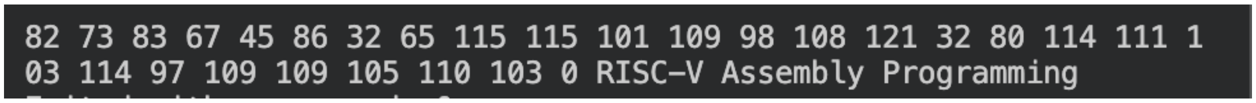
VS Code is an integrated development environment (IDE) where Venus and RISC-V can be accessed and programmed. “RISC-V is an open-source instruction set architecture used to develop custom processors for a variety of applications, from embedded designs to supercomputers." according to Synopsys.. "Terminal is an app for advanced users and developers that lets you communicate with the Mac operating system using a command line interface (CLI)." (Apple). Using VSCode, a programmer can use RISC-V to take in inputs, perform operations, and return outputs. Terminal provides a place to run programs that can output errors, what's causing them, what line, and which column to better assist programmers.

Methodology/Procedure

The procedure starts with writing a program to test and learn more about how the program loops through strings and returns a certain value. Next, the code is reused to run through the entire string and count the number of characters in the inputted string. Then, the code is reused again to the number of spaces in a string using a command such as "li t0, ' '". This program uses this string to teach students what looping in strings looks like and how we can use this to count, compare, and more. Reading an string through a loop, calculating values such as count and comparison using commands such as "bne" and "beq" with ASCII values are coded into the program. Finally, a last program is written to compare two strings for differences. The input strings vary with only one letter being uppercase and lowercase.

Documentation

Part One Output:



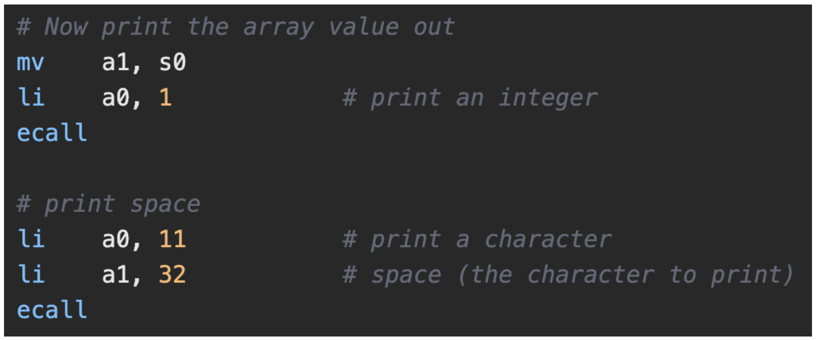
```
82 73 83 67 45 86 32 65 115 115 101 109 98 108 121 32 80 114 111 1
03 114 97 109 109 105 110 103 0 RISC-V Assembly Programming
```

Questions:

Part One:

1. **This code does more than just copy a string. What changes must be made to only copy a string and print the copy?**

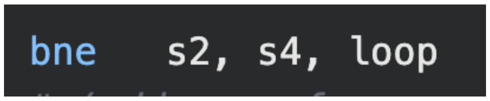
   The given code is printing ASCII values during the copy loop and printing the values. In order to change the code to only copy a string and print the copy, it needs to be modified by removing the lines that print the ASCII values inside the loop and only keeping the copy loop.



```
# Now print the array value out
mv    a1, s0
li    a0, 1            # print an integer
ecall

# print space
li    a0, 11           # print a character
li    a1, 32           # space (the character to print)
ecall
```

2. **What does the bne instruction do in the loop?**

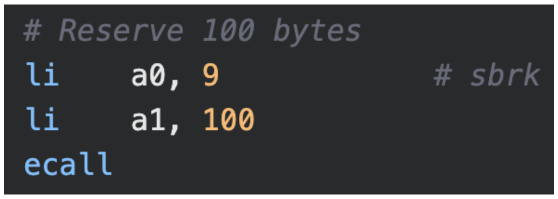
   The bne instruction is using "bne" as a *branch if not equal* to compare s2 and s4. If the two values are not equal, the program branches back to the loop and continues the copy; if they are equal, it exits the loop.



```
bne    s2, s4, loop
```

3. **What would happen if the copy buffer is not allocated enough space to hold the entire string?**

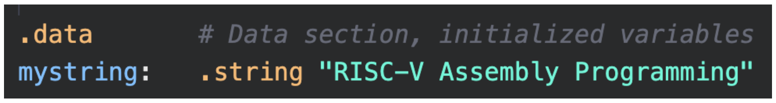
   If the copy buffer is not allocated enough space to hold the entire string with sbrk, the loop will continue to write. Doing this might overwrite other data–which could lead to the program crashing.



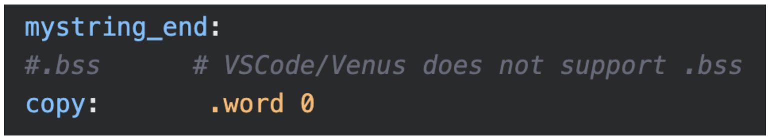
```
# Reserve 100 bytes
li     a0, 9            # sbrk
li     a1, 100
ecall
```

4. **How does the code ensure that the copied string (copy) is null terminated?**

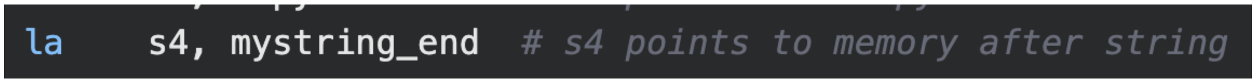
   The following lines work to indicate that the program ends automatically when .string adds "RISC-V Assembly Programming". The line "la s4, mystring_end" is what makes the loop stop after copying the null byte and null terminated.



```
.data        # Data section, initialized variables
mystring:    .string "RISC-V Assembly Programming"
```



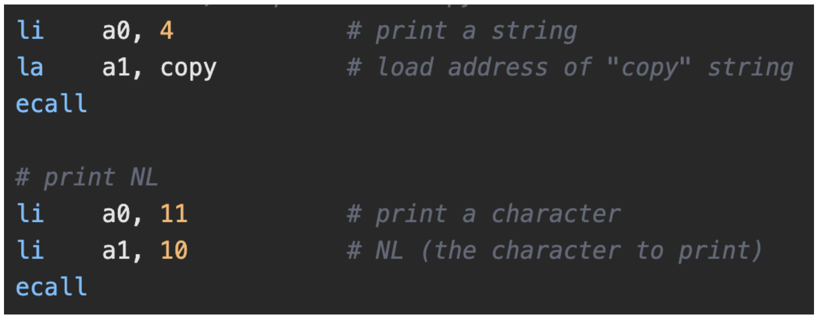
```
mystring_end:
#.bss       # VSCode/Venus does not support .bss
copy:        .word 0
```



```
la    s4, mystring_end  # s4 points to memory after string
```

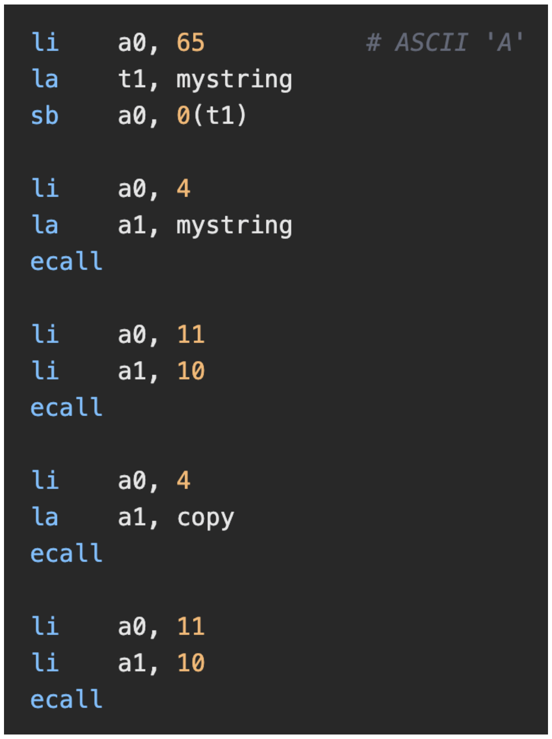
5. **How do we know that this works? What modification could you make to convince someone that it does actually make a copy and is not simply printing the same thing twice?**

The given string can be modified after copying and printing both the original and copy version. “mystring” can be changed from:



```
li    a0, 4             # print a string
la    a1, copy          # load address of "copy" string
ecall

# print NL
li    a0, 11            # print a character
li    a1, 10            # NL (the character to print)
ecall
```

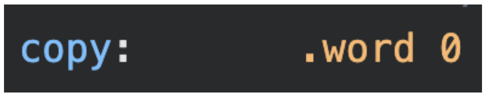
to the following section of code to convince someone that it actually does make a copy:



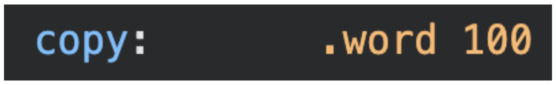
```
li    a0, 65          # ASCII 'A'
la    t1, mystring
sb    a0, 0(t1)

li    a0, 4
la    a1, mystring
ecall

li    a0, 11
li    a1, 10
ecall

li    a0, 4
la    a1, copy
ecall

li    a0, 11
li    a1, 10
ecall
```

6. **Why do we have the line copy: .word 0 instead of copy: .word 100 if we want "copy" to point to 100 bytes?**

The reason we have the line copy: .word 0 instead of copy:.word 100 if we want “copy” to print to 100 bytes is because .word 0 is not allocating 100 bytes, .word 100 is; in this case, it is working with sbrk as a placeholder.Using .word 100 would initialize the specific word with 100 as a value and not allocate 100 bytes of space.



```
copy:         .word 0
```



```
copy:         .word 100
```

Part Two:

1. **Why do we use "lb" to get a character, and not "lw"?**

   We use “lb” to get a character and not “lw” because lb” loads one byte which consists of 8 bits from memory to register. “lw”, on the other hand, loads 4 bytes consisting of 32 bits. This difference is why we use “lb” because strings are byte arrays.

2. **If the string were to contain non-ASCII characters or multi-byte characters, how would that affect the length reported?**

   If the string were to contain non-ASCII characters or multi-byte characters, this would affect the length reported wrong. Since ASCII characters are one to four bytes and the subroutine is counting bytes, having non-ASCII or multi-byte characters report back wrong values to count.

3. **If you wanted to use different registers for the values passed to and from the subroutine, what changes would you have to make?**

   If I wanted to use different registers for the values passed to and from the subroutine, some changes that need to be made are to break the calling convention. This way, the different registers can still seamlessly process and run the program as intended.

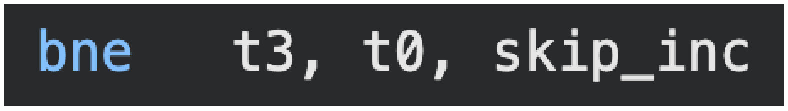
4. **Suppose that the string ends with a space. Would it be obvious to the user that the number reported is correct?**

   If the string ends with a space, it would be obvious to the user that the number reported is correct because the space in between the quotation marks in the code will show it clearly.

Part Three:

1. **Suppose that your friend sees your program to count spaces, and tells you that you should use blt instead of the branch command that you use at the "If not, skip around the increment" part. Would that work? Explain.**

   Using “blt’ instead of the branch command that I use at the “If not, skip around the increment” part would not work. This is because the current code “bne t3, t9, skip_inc” is working to skip incrementing if the character in t3 is not a space (denoted by t0). Using “blt” meaning “branch if less than” would increment the character only if the character in t3 is less than the space in t0.



```
bne   t3, t0, skip_inc
```

2. **Describe what the instructions beq and bne do.**

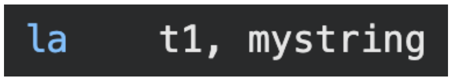
The instruction “beq” is “branch if equal”. It compares the first and second arguments, and if they are both equal, it jumps to the third part. For example, “beq t1, t0, exit” would compare t1 and t0, and if they are equal, it would jump to “exit”. “bne” means “branch if not equal” which would compare the first and second arguments, and if they are not equal, it jumps to the third part. For example, “bne t1, t0, exit” would compare t1 and t0, and if they are not equal, it would jump to “exit”.

3. **What if the string has two or more spaces in a row? Does the code still work?**

   If the string has two or more spaces in a row, the code would still work because the count is counting one character at a time. While looping through the string, each space would be successfully counted.

4. **What if the string begins with a space? Does the code still work?**

   If the string begins with a space, the could would still work because the loop checks every character of the string and compares it. The space would still work and it would still increment the counter accordingly.

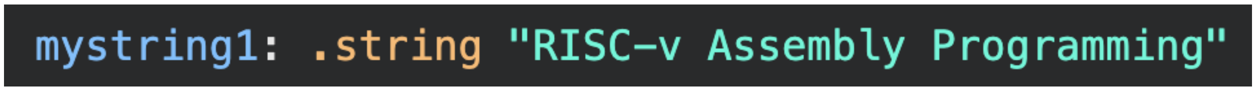


```
la    t1, mystring
```

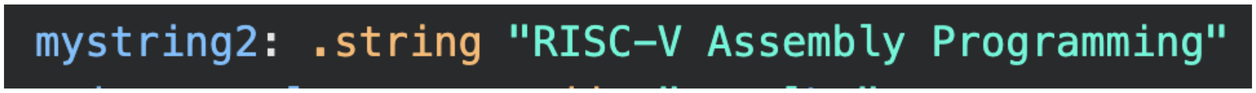
Part Four:

1. **Are the original two given strings the same? Why or why not?**

   No, the original two given strings are not the same. While the spelling and punctuation are exact, the letter “v” is lowercase in the first string and uppercase in the second string. According to IBM.com, the ASCII value for lowercase “v” is 86 while the ASCII value for uppercase “V” is 118. This difference is important in this program because the values in the code are based using ASCII values.

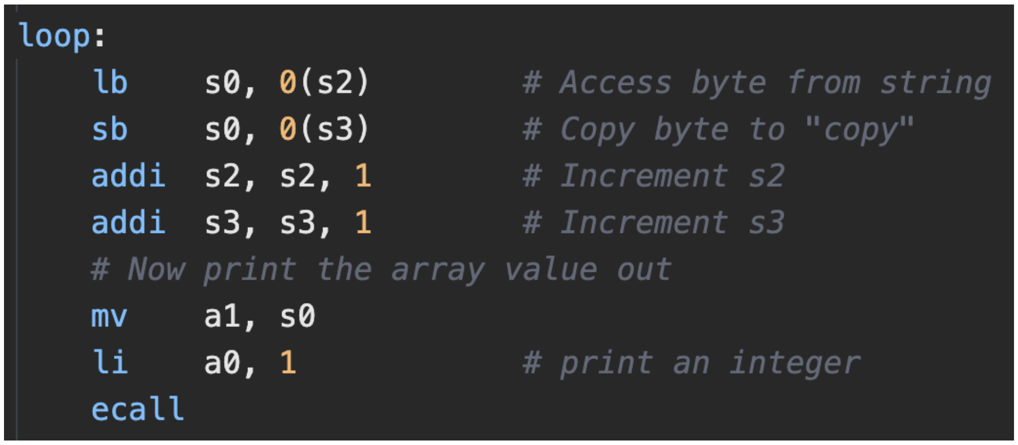


```
mystring1: .string "RISC-v Assembly Programming"
```

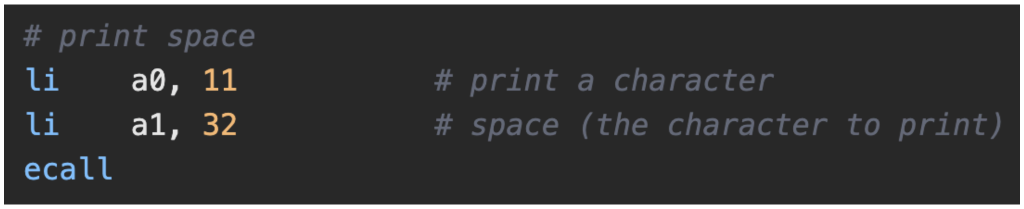


```
mystring2: .string "RISC-V Assembly Programming"
```

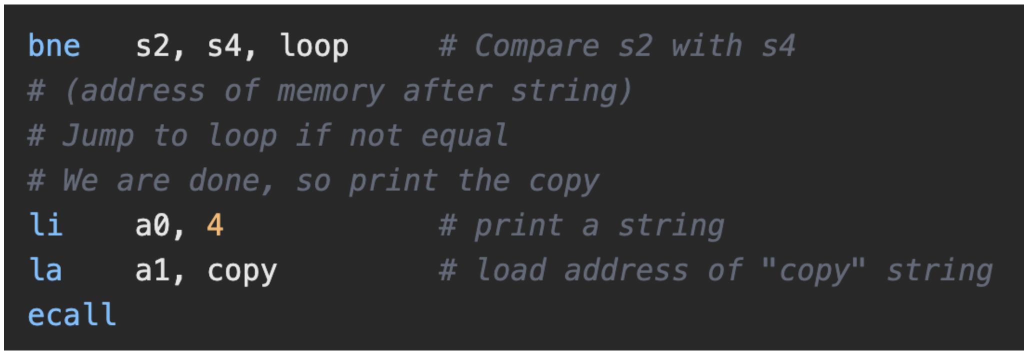
Key Code Observations



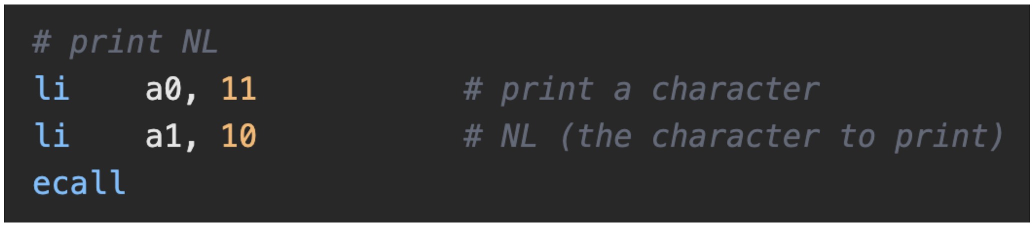
```
loop:
    lb    s0, 0(s2)         # Access byte from string
    sb    s0, 0(s3)         # Copy byte to "copy"
    addi  s2, s2, 1         # Increment s2
    addi  s3, s3, 1         # Increment s3
    # Now print the array value out
    mv    a1, s0
    li    a0, 1             # print an integer
    ecall
```



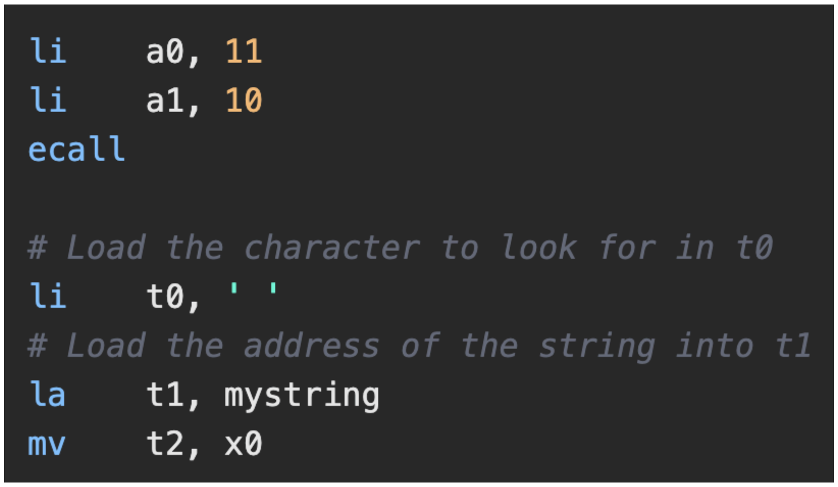
```
# print space
li    a0, 11          # print a character
li    a1, 32          # space (the character to print)
ecall
```



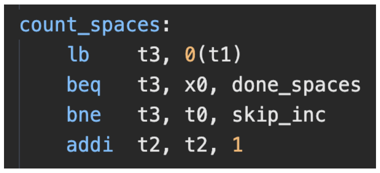
```
bne   s2, s4, loop      # Compare s2 with s4
# (address of memory after string)
# Jump to loop if not equal
# We are done, so print the copy
li    a0, 4             # print a string
la    a1, copy          # load address of "copy" string
ecall
```



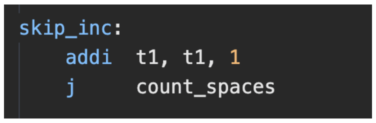
```
# print NL
li    a0, 11          # print a character
li    a1, 10          # NL (the character to print)
ecall
```



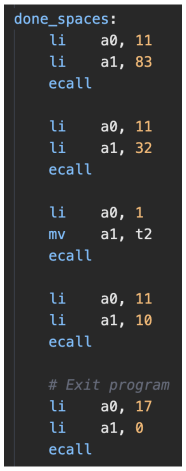
```
li    a0, 11
li    a1, 10
ecall

# Load the character to look for in t0
li    t0, ' '
# Load the address of the string into t1
la    t1, mystring
mv    t2, x0
```



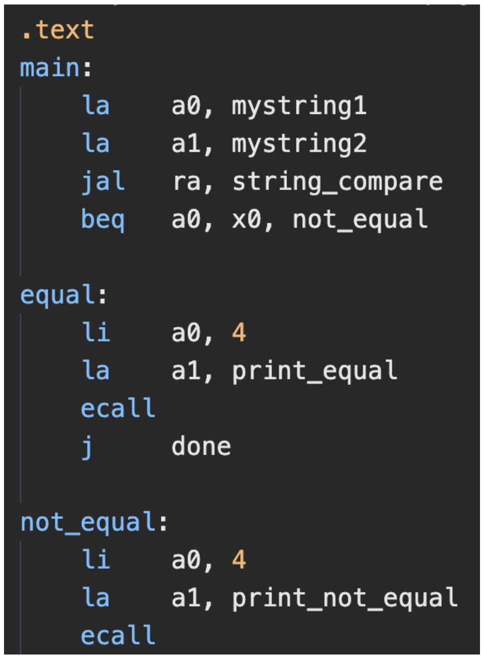
```
count_spaces:
    lb    t3, 0(t1)
    beq   t3, x0, done_spaces
    bne   t3, t0, skip_inc
    addi  t2, t2, 1
```



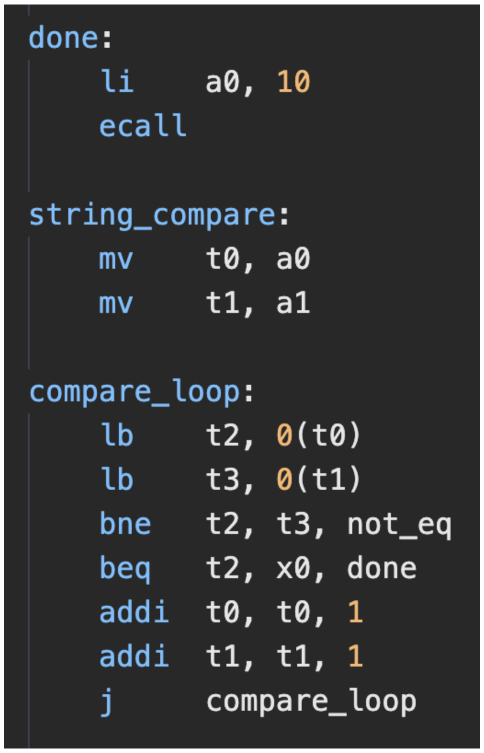
```
skip_inc:
    addi  t1, t1, 1
    j     count_spaces
```



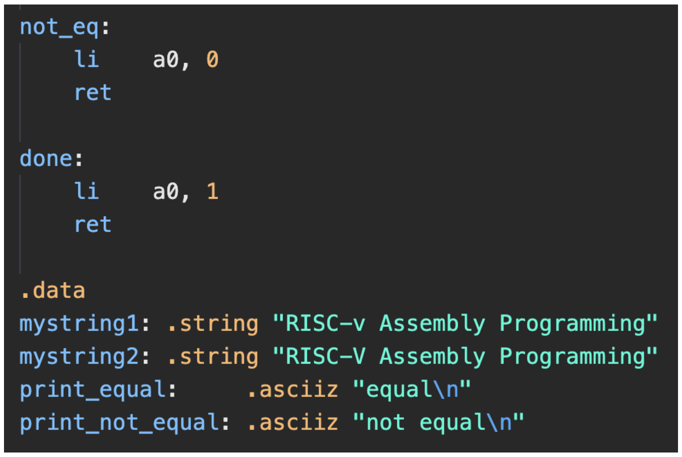
```
done_spaces:
    li    a0, 11
    li    a1, 83
    ecall

    li    a0, 11
    li    a1, 32
    ecall

    li    a0, 1
    mv    a1, t2
    ecall

    li    a0, 11
    li    a1, 10
    ecall

    # Exit program
    li    a0, 17
    li    a1, 0
    ecall
```



```
.text
main:
    la    a0, mystring1
    la    a1, mystring2
    jal   ra, string_compare
    beq   a0, x0, not_equal

equal:
    li    a0, 4
    la    a1, print_equal
    ecall
    j     done

not_equal:
    li    a0, 4
    la    a1, print_not_equal
    ecall
```



```
done:
    li    a0, 10
    ecall

string_compare:
    mv    t0, a0
    mv    t1, a1

compare_loop:
    lb    t2, 0(t0)
    lb    t3, 0(t1)
    bne   t2, t3, not_eq
    beq   t2, x0, done
    addi  t0, t0, 1
    addi  t1, t1, 1
    j     compare_loop
```



```
not_eq:
    li    a0, 0
    ret

done:
    li    a0, 1
    ret

.data
mystring1: .string "RISC-v Assembly Programming"
mystring2: .string "RISC-V Assembly Programming"
print_equal:     .asciiz "equal\n"
print_not_equal: .asciiz "not equal\n"
```